Online Balancing of Range-Partitioned Data with Applications to Peer-to-Peer Systems

Prasanna Ganesan Mayank Bawa Hector Garcia-Molina

Stanford University
Stanford, CA 94305
{*prasannag, bawa, hector*}*@cs.stanford.edu*

Abstract

We consider the problem of horizontally partitioning a dynamic relation across a large number of disks/nodes by the use of range partitioning. Such partitioning is often desirable in large-scale parallel databases, as well as in peer-to-peer (P2P) systems. As tuples are inserted and deleted, the partitions may need to be adjusted, and data moved, in order to achieve storage balance across the participant disks/nodes. We propose efficient, asymptotically optimal algorithms that ensure storage balance at all times, even against an adversarial insertion and deletion of tuples. We combine the above algorithms with distributed routing structures to architect a P2P system that supports efficient range queries, while simultaneously guaranteeing storage balance.

1 Introduction

The problem of partitioning a relation across multiple disks has been studied for a number of years in the context of parallel databases. Many shared-nothing parallel database systems use range partitioning to decluster a relation across the available disks for performance gains [7, 9, 23]. For example, transactions in OLTP systems often access tuples associatively, i.e., all tuples with a specific attribute value, or a small range of values. Range partitioning ensures that a transaction requires data only from a single disk (most of the time), thus enabling inter-query parallelism and near-linear speed-up [10].

A well-known concern in range partitioning is *skew*, where only a few partitions (disks/nodes) are involved in the execution of most queries. Skew can be classified into (a) *data skew*, where data may be unequally distributed across the partitions, and (b) *execution skew*, where data accesses may not be uniform across the partitions [10]. As the relation evolves over time, or as workloads change, both data and execution skew pose a serious problem.

Today's database systems put the onus on administrators to monitor performance and re-partition data whenever skew becomes "too large", an approach fraught with difficulties. In contrast, we consider *online load-balancing* solutions, which dynamically move data across nodes and avoid skew *at all times*. Online load-balancing promises three major advantages over periodic manual repartitionings: (a) a consistently efficient 24/7 operation by eliminating performance degradation between, and system hiccups during, manual repartitioning; (b) a simplified control panel by eliminating partition configuration from the administrator's list of chores; and (c) a smaller cost especially in systems with a high degree of parallelism, where even a few inserts/deletes may cause a large skew.

Skew can be characterized by the *imbalance ratio* σ defined as the ratio of the loads of the largest and smallest partitions in the system. In order to ensure that σ is small, data may have to be moved from one disk/node to another as the relation grows or shrinks. Thus a key requirement for a load balancing algorithm is to minimize the number of tuples moved in order to achieve a desired σ.

Summary of Results In this paper, we focus on algorithms for eliminating data skew to achieve storage balance, although our algorithms can be easily generalized to handle execution skew as well. Our load-balancing algorithms guarantee that σ is always bounded by a *small constant* c. The bound c is, in fact, a tunable parameter that can be set to values as low as 4.24. Moreover, each insert or delete of a tuple is guaranteed to require just an (amortized) *constant* number of tuple movements, even against an *adversarial* sequence of inserts and deletes. Thus, our algorithms offer storage balance at all times, against all data distributions, while ensuring that the overhead is asymptotically optimal, and often much less than that of periodic repartitioning.

Proceedings of the 30th VLDB Conference,
Toronto, Canada, 2004

Application to P2P Systems Our online load balancing algorithms are motivated by a new application domain for range partitioning: peer-to-peer (P2P) systems. P2P systems store a relation over a large and dynamic set of nodes, and support queries over this relation. Many current systems, known as Distributed Hash Tables (DHTs) [17, 18, 22], use hash partitioning to support point queries.

There has been considerable recent interest in developing P2P systems that can support efficient range queries [3, 4, 20]. For example, in a P2P auction system, a node might desire a list of all computers for sale, priced between $400 and $600. Similarly, in a P2P web cache, a node may request (pre-fetch) all pages with a specific URL prefix. It is well-known [5] that hash partitioning (and hence a DHT) is inefficient for answering such *ad hoc* range queries, motivating a search for new networks that can support efficient range partitioning in the face of data and execution skews.

The P2P domain throws up its own challenges for range-partitioning and load balancing. P2P systems are dynamic in that nodes may arrive and depart at will. The scheme thus needs to ensure that balance is achieved on such a dynamic set of nodes. In addition, P2P systems are decentralized, necessitating the design of distributed data structures for maintaining partition information. We show how to enhance our online load-balancing algorithm with overlay-network structures to architect a new P2P system whose performance is asymptotically identical to that of DHTs, but with the advantage of enabling efficient range queries.

Organization We define the online load-balancing problem for parallel databases in Section 2. We present our load-balancing algorithm and analyze it in Section 3. We adapt our algorithm to a P2P setting in Section 4. We experimentally evaluate our algorithms in Section 5. We discuss related work in Section 6.

2 Problem Setup and Basic Operations

In this section, we define a simple abstraction of a parallel database, discuss a cost model for load balancing in this context, and define two basic operations used by load-balancing algorithms. We defer a discussion of our model of P2P systems to Section 4.

2.1 Setup

We consider a relation divided into n range partitions on the basis of a key attribute, with partition boundaries at $R_0 \leq R_1 \leq \ldots \leq R_n$. Node N_i manages the range $[R_{i-1}, R_i)$, for all $0 < i \leq n$. When $R_{i-1} = R_i$, N_i is said to manage the *empty* partition $[R_{i-1}, R_i)$. Nodes managing adjacent ranges are said to be *neighbors*. We let $L(N_i)$ denote the *load* at N_i defined to be the number of tuples stored by N_i. We assume a central site has access to the range-partition information $[R_0, R_1, \ldots, R_n]$ and directs each query, insert and delete to the appropriate node(s). [1]

[1] Alternatively, one could replicate this range information across all nodes. If broadcast across nodes is as cheap as unicast, this approach is practically identical to the centralized approach.

Each insert or delete of a tuple is followed by an execution of the load-balancing algorithm which may possibly move data across nodes. The load-balancing algorithms we consider are *local* in that the algorithm executes only on the node at which the insert or delete occurs. For now, we ignore concurrency control issues (see Section 3.4), and consider only the equivalent serial schedule of inserts and deletes, interleaved with the executions of the load-balancing algorithm.

Imbalance Ratio A load-balancing algorithm guarantees an imbalance ratio σ if, after the completion of every insert or delete operation and its corresponding load-balancing step, $\max_i L(N_i) \leq \sigma \min_i L(N_i) + c_0$, for some fixed constant c_0. As is conventional, we have defined σ as the *asymptotic* ratio between the largest and smallest loads.

2.2 Costs of Load Balancing

Data Movement All load-balancing algorithms will need to move data from one node to another in order to achieve balance. We use a simple linear cost model, where moving one tuple from any node to any other node costs one unit. Such a model reasonably captures both the network-communication cost of transferring data, as well as the cost of modifying local data structures at the nodes.

Partition Change Data movement is accompanied by a change in the partition boundaries. The central site needs to be informed to enable it to correct its partition information $[R_0, R_1, \ldots, R_n]$. Notice that the movement of a tuple may cause a change in at most *one* partition boundary, resulting in at most one update message to the central site. We can thus absorb this cost into the data movement cost itself.

Load Information Finally, the load-balancing algorithm that executes locally at a node may require non-local information about the load at other nodes. For now, we assume that the central site keeps track of the load at each node, thus requiring each node to inform the site after a successful insert, delete or data movement. A node that needs load information can simply contact the central site at any time to obtain it. We can thus absorb this cost into the cost of tuple insert, delete and movement as well.

In summary, we measure the cost of a load-balancing algorithm simply as the number of tuples moved by the algorithm per insert or delete. Our interest is in the *amortized* cost per insert or delete, for *adversarial* (worst-case) sequences of insertions and deletions. The amortized cost of an insert or delete is said to be c if, for *any* sequence of t tuple inserts and deletes, the total number of tuples moved is at most tc.

Problem Statement Develop a load balancing algorithm which guarantees a constant imbalance ratio σ with low amortized cost per tuple insert and delete.

We will show that it is possible to achieve a constant σ while ensuring that the amortized cost per insert and delete is also a constant. Such an algorithm is asymptotically optimal since, for any load-balancing algorithm, there exist se-

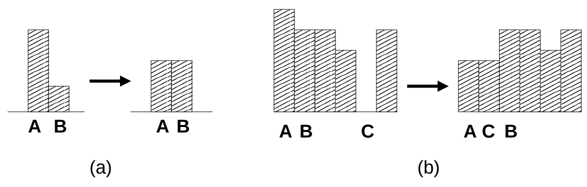

Figure 1: (a) NBRADJUST involving A and B and (b) REORDER involving A and C. The height of a bar represents the load of the corresponding node.

quences of t operations that require $\Omega(t)$ tuple movements to ensure load balance.

2.3 "Universal" Load-Balancing Operations

What operations can be used to perform load balancing? An intuitive operation is as follows: when a node becomes responsible for too much data, it can move a portion of its data to its neighbor and thus attempt to balance out the load. We call such an operation NBRADJUST which is illustrated in Figure 1(a) and defined below.

NBRADJUST *A pair of neighboring nodes N_i and N_{i+1} may alter the boundary R_i between their ranges by transferring data from one node to the other.* [2]

A load-balancing algorithm can be devised based on just this operation, e.g. [11, 15]. However, such an algorithm is provably expensive as we show in the following theorem.

Theorem 1. *Any load-balancing algorithm, deterministic or randomized, that uses only* NBRADJUST *and guarantees a constant imbalance ratio σ, has amortized cost $\Omega(n)$ per insert and delete.*

Proof. Say a load-balancing algorithm using only NBRADJUST, guarantees that the maximum load $L_{max} \leq \sigma L_{min} + c_0$, for some constants σ and c_0.

Consider the initial state of the system where each of the nodes $N_1, N_2, \ldots N_n$ all have zero load. Now consider an adversary producing a sequence of $t = c_0^2 + n^2$ insertions $I_1, I_2, \ldots I_t$ with corresponding keys $k_1 > k_2 > \ldots > k_t$.

After the first c_0^2 insertions, node N_1 must contain key $k_{c_0^2}$, since its load would be zero otherwise, causing an imbalance ratio violation. All subsequent insertions all occur at node N_1, since the last inserted key is always the smallest key inserted so far.

At the end of all tuple insertions, there are more than n^2 tuples in all. Since σ is a constant, each node must have load at least n/c, for some constant c. Thus, each node receives at least $c'n$ tuples that it did not possess after the initial phase of c_0^2 insertions, for some constant c'. Since n^2 of the tuples were inserted at node N_1, and all tuple transfers may occur only between neighboring nodes, the only means for node N_k to receive tuples is from N_{k-1}. Thus, the total number of tuple movements necessary for node N_k to receive its load is at least $c'nk$. Summing over all $k > 1$, the total data movements necessary is at least $\Theta(n^3)$. Since the total number of insertions is n^2, it follows that the amortized cost of an insertion is $\Omega(n)$.

□

[2] In the extreme case when N_i takes over the entire range, $[R_{i-1}, R_{i+1})$, N_{i+1} is assigned the empty range $[R_{i+1}, R_{i+1})$.

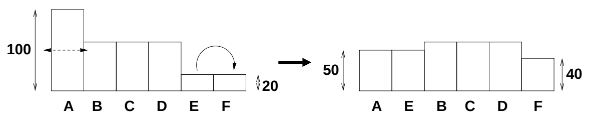

Figure 2: The cost of load balancing using REORDER is 70 while using successive NBRADJUST operations costs 250.

The above theorem shows that any algorithm that uses only NBRADJUST would incur a cost per insert that is at least *linear* in the number of nodes. In contrast, our goal is to achieve a *constant* cost per insert. The key to efficient load balancing lies in a second operation, REORDER, illustrated in Figure 1(b) and defined below.

REORDER *A node N_i with an empty range $[R_i, R_i)$ changes its position and splits the range $[R_j, R_{j+1})$ managed by a node N_j: N_j now manages range $[R_j, X)$ while N_i takes over $[X, R_{j+1})$ for some value of X, $R_j \leq X \leq R_{j+1}$. The nodes are re-labeled appropriately.*

EXAMPLE 2.1. *Consider the scenario shown in Figure 2, where node A has* 100 *tuples, the next three nodes (B, C, D) have* 60 *tuples each, while the last two (E, F) have* 20 *tuples each. The least expensive scheme to improve load balance while preserving key ordering is to transfer all* 20 *tuples from E to F, and then use* REORDER *to split the load of A between A and E. The cost of such a scheme is* 70 *tuple movements; in contrast, a* NBRADJUST*-based balancing requires* 250 *tuple movements.* ■

It turns out that the REORDER operation is not only necessary, but also sufficient for efficient load balancing. In fact, we show below that the operations NBRADJUST and REORDER are *universal* in that they can together be used to efficiently implement any load-balancing algorithm.

Theorem 2. *Given a load-balancing algorithm A, it is possible to construct a new algorithm $\widehat{A}$ that uses only the* NBRADJUST *and* REORDER *operations such that, for any sequence of t inserts and deletes,*
(a) Both A and $\widehat{A}$ achieve identical load distribution.
(b) The cost of $\widehat{A}$ is at most the cost of A.

3 Algorithms for Load Balancing

Consider the following approach for load balancing: a node attempts to shed its load whenever its load increases by a factor δ, and attempts to gain load when it drops by the same factor. Formally, we consider an infinite, increasing geometric sequence of thresholds $T_i = \lfloor c\delta^i \rfloor$, for all $i \geq 1$ and some constant c. When a node's load crosses a threshold T_j, the node initiates a load-balancing procedure. We call such an approach the *Threshold Algorithm*.

3.1 The Doubling Algorithm

We start with the special case $\delta = 2$, and the thresholds $T_i = 2^{i-1}$. We begin by considering tuple insertions. Every time a node's load[3] increases to a value $T_i + 1$, the node initiates ADJUSTLOAD specified in Procedure 1.

The load-balancing procedure ADJUSTLOAD is quite simple. When node N_i's load increases beyond a threshold, it first (lines 3-6) attempts to perform NBRADJUST with its lightly-loaded neighbor, say N_{i+1}, by averaging out its load with N_{i+1}. If both neighbors have high load (more than half that of N_i), N_i attempts to perform REORDER with the globally least-loaded node N_k (lines 8-12). If N_k's load is small enough (less than a quarter of N_i), N_k sheds all its data to $N_{k\pm1}$, and takes over half the load of N_i. If N_i is unable to perform either NBRADJUST or REORDER, N_i concludes that the system load is indeed balanced and performs no data movement.

Note that when N_i initiates either a NBRADJUST or REORDER, there is a corresponding recursive invocation of ADJUSTLOAD at node N_{i+1} or $N_{k\pm1}$ respectively. Frequently, these recursive invocations do not necessitate any further data movement; even if data movement is necessary, we can show that such data movement would utilize only NBRADJUST. Similarly, there is also a recursive invocation of ADJUSTLOAD at node N_i itself (line 6); this invocation is necessary only in one special case – when ADJUSTLOAD is being executed at $N_{k\pm1}$ after a REORDER – and is also guaranteed to utilize only NBRADJUST.

Procedure 1 ADJUSTLOAD(Node N_i) {On Tuple Insert}

```
1: Let $L(N_i) = x \in (T_m, T_{m+1}]$.
2: Let $N_j$ be the lighter loaded of $N_{i-1}$ and $N_{i+1}$.
3: if $L(N_j) \leq T_{m-1}$ then {Do NBRADJUST}
4:     Move tuples from $N_i$ to $N_j$ to equalize load.
5:     ADJUSTLOAD($N_j$)
6:     ADJUSTLOAD($N_i$)
7: else
8:     Find the least-loaded node $N_k$.
9:     if $L(N_k) \leq T_{m-2}$ then {Do REORDER}
10:        Transfer all data from $N_k$ to $N = N_{k\pm1}$.
11:        Transfer data from $N_i$ to $N_k$, s.t. $L(N_i) = \lceil x/2 \rceil$ and
           $L(N_k) = \lfloor x/2 \rfloor$.
12:        ADJUSTLOAD($N$)
13:        {Rename nodes appropriately after REORDER.}
14:    end if
15: end if
```

Deletions are handled in a symmetric fashion. When a node's load drops to a threshold $T_j = 2^j$, it first attempts NBRADJUST with a neighbor, if the neighbor's load is larger than $T_{j+1} = 2^{j+1}$. Otherwise, it attempts to REORDER itself and split the highest-loaded node N_k in the system, if N_k's load is more than T_{j+2}.

We will show later that the Doubling Algorithm ensures that $\sigma = 8$, while the amortized cost of tuple insert and delete is constant. However, it is possible to reduce σ further by generalizing this Doubling Algorithm.

[3]For technical reasons, we define $L'(N) = T_1 + L(N)$, and use L' as the node load. Note that the same guarantees on σ hold when using either L or L'; for notational convenience, we let L denote this new definition of load.

3.2 The General Threshold Algorithm

The Doubling Algorithm set $\delta = 2$ and triggered load balancing when a node's load changed by a factor 2 to obtain $\sigma = 8$. A natural question, then, is to ask whether the algorithm generalizes to other values of δ, and whether it is possible to obtain a better σ by using a smaller δ value.

The Doubling algorithm generalizes to allow δ to be any real number greater than or equal to the golden ratio $\phi = (\sqrt{5}+1)/2 \simeq 1.62$. For any real number $\delta \geq \phi$, we may define a general *Threshold Algorithm* as follows: We define a threshold sequence of $T_i = \lfloor c\delta^i \rfloor$, for an appropriately chosen constant $c > 0$. Each node is required to execute Procedure ADJUSTLOAD, every time its load crosses a threshold. This Threshold Algorithm guarantees $\sigma = \delta^3$ with a constant cost per tuple insert and delete.

The Fibbing Algorithm: An extreme of the general Threshold Algorithm arises when $\delta = \phi$, for which we may define a variant called the *Fibbing Algorithm*. This algorithm defines the set of thresholds T_i to be the Fibonacci numbers (with $T_1 = 1$ and $T_2 = 2$). As we prove in Section 3.3, the Fibbing Algorithm guarantees an imbalance ratio of $\phi^3 \simeq 4.24$.

3.3 Analysis

We now present an analysis of the Threshold algorithm (and the Fibbing algorithm), both in terms of the guaranteed imbalance ratio, and in terms of the cost of insert and delete. Our analysis relies on seven properties of the threshold sequence that is satisfied both by Fibonacci numbers, and by threshold sequences of the form $T_i = \lfloor c\delta^i \rfloor$, allowing the same analysis to apply to both the Threshold algorithm and the Fibbing algorithm. We summarize these properties in the following lemma.

Lemma 1. *If $T_i = \lfloor c\delta^i \rfloor$, for a suitably large c and $\delta \geq \phi$, the following properties hold for all $r \geq 1$. The same properties hold if T_i is the i^{th} Fibonacci number ($T_i = [c\delta^i]$, with $c = \phi/\sqrt{5}$ and $\delta = \phi$).*
(a) $\lfloor (T_r + T_{r+2})/2 \rfloor \geq T_{r+1}$
(b) $\lceil (T_r + T_{r+1} + 1)/2 \rceil \leq T_{r+1}$
(c) $T_r + T_{r+1} \leq T_{r+2}$
(d) $\lceil (T_r + 1)/2 \rceil \leq T_r$
(e) $\lfloor (T_{r+2} + 1)/2 \rfloor > T_r$
(f) $T_{r+k} + 1 \geq \delta^k T_r \geq T_{r+k} - C$, *where $C = 1$ if T_i is the i^{th} Fibonacci number, and $C = \delta^k$ otherwise, for all integers $k > 0$.*
(g) $\lfloor (T_1 + T_2 + 1)/2 \rfloor > T_1$

Definition 3.1. *For any node N, define $I(N) = r$ if and only if $L(N) \in (T_{r-1}, T_r]$, i.e., N's load is in the r^{th} geometric interval.*

Theorem 3. *The following invariants hold after any sequence of inserts and deletes for the Threshold (and Fibbing) algorithm:*

(a) NBRBALANCE: *For any pair of neighbors N_i and N_{i+1}, $I(N_i) \leq I(N_{i+1}) + 1$.*

(b) GLOBALBALANCE: *For any pair of nodes N_i and N_j, $I(N_i) \leq I(N_j) + 2$.*

Before proving the above theorem, we first establish some lemmas on the properties of the NBRADJUST and REORDER, as well as the execution of ADJUSTLOAD.

Lemma 2. *If $I(N_i) = r + 2$ and $I(N_{i+1}) = r$, then* NBRADJUST *between N_i and N_{i+1} ensures that $I(N_i) = I(N_{i+1}) \geq r + 1$.*

Proof. If $r > 1$, we know $L(N_i) > T_{r+1}$ and $L(N_{i+1}) > T_{r-1}$. The new loads at N_i and N_{i+1} are at least $l = \lfloor(\lfloor T_{r+1}\rfloor + 1 + \lfloor T_{r-1}\rfloor + 1)/2\rfloor$. By Lemma 1(a), we know that $l \geq 1 + \lfloor T_{r-1}\rfloor > T_{r-1}$.

If $r = 1$, then $L(N_{i+1}) = T_1 = \lceil T_1\rceil$ (by definition, since all loads are at least $\lceil T_1\rceil$), and $L(N_i) \geq \lfloor T_2\rfloor + 1$. By Lemma 1(g), the new loads at N_i and N_{i+1} are greater than T_1. □

Lemma 3. *Consider a state of the system where both* NBRBALANCE *and* GLOBALBALANCE *invariants hold. If a tuple insert now causes a violation of* NBRBALANCE, *the consequent execution of* ADJUSTLOAD *will ensure both* NBRBALANCE *and* GLOBALBALANCE.

Proof. Consider a tuple insert at node N_i. By definition, no NBRBALANCE violation arises unless $L(N_i)$ crosses a threshold. Say $L(N_i)$ crosses threshold T_x. There may then be a violation of NBRBALANCE between N_i and either or both of its neighbors. (There may also be a GLOBALBALANCE violation involving N_i.) In this case, N_i executes a NBRADJUST by Procedure ADJUSTLOAD.

W.l.o.g., say N_i performs NBRADJUST with N_{i+1}. First, observe that there are no GLOBALBALANCE violations after this NBRADJUST, by Lemma 1(b). After this NBRADJUST, ADJUSTLOAD is recursively invoked on N_{i+1}, which may cause N_{i+1} to perform NBRADJUST with N_{i+2}, and trigger N_{i+2} into executing ADJUSTLOAD. This process continues until we reach a node N_{i+k} that does not perform a NBRADJUST, or we reach N_n.

We show that this sequence of NBRADJUST operations ensures that all NBRBALANCE conditions are satisfied. (Finally, there are recursive calls to ADJUSTLOAD in line 6, which do not perform any data movement since there are no violations of NBRBALANCE or GLOBALBALANCE.)

Let $L_j(N)$ represent the load at node N after the j^{th} NBRADJUST operation, and $I_j(N) = I(N)$ after the j^{th} NBRADJUST operation. (L_0 is the load before any NBRADJUST operations take place.) The j^{th} NBRADJUST operation occurs between nodes N_{i+j-1} and N_{i+j}. Thus, the load of N_{i+j-1} remains unchanged after the j^{th} operation.

We will show by induction that, after $j > 0$ NBRADJUST operations,

1. $I_j(N_{i+k}) \leq x$ for all $0 \leq k \leq j$.
2. The only NBRBALANCE violation may be between N_{i+j} and N_{i+j+1}.

Base Case: We show the above properties for $j = 1$. Initially, the only NBRBALANCE violations may be at (N_{i-1}, N_i) and/or (N_i, N_{i+1}). Recall that $I_0(N_i) = x + 1$ and, since there is a NBRBALANCE violation, $I_0(N_{i+1}) = x - 1$. From the GLOBALBALANCE invariant, we deduce $I_0(N_{i-1}) \leq x + 1$.

After the first NBRADJUST operation, we know by Lemma 2 that $I_1(N_i) = I_1(N_{i+1}) \geq x$. Also, $L_1(N_{i-1}) = L_0(N_{i-1})$, thus showing that neither pair (N_{i-1}, N_i) nor (N_i, N_{i+1}) constitute a NBRBALANCE violation. Since only the loads of N_i and N_{i+1} were affected by this operation, the only possible NBRBALANCE violation is between N_{i+1} and N_{i+2}. It is also clear that $I_1(N_i) = I_1(N_{i+1}) \leq x$ (by Lemma 1(d)).

Induction Step: Assume, by induction, that after j NBRADJUST operations, $I_j(N_{i+j}) \leq x$, and the only possible NBRBALANCE violation is at (N_{i+j}, N_{i+j+1}). If there is no such violation, we are done and ADJUSTLOAD terminates. If there is a violation, a NBRADJUST takes place between N_{i+j} and N_{i+j+1}.

GLOBALBALANCE assures us that $I_0(N_{i+j+1}) = I_j(N_{i+j+1}) \geq x - 2$. Since there is a NBRBALANCE violation, we may deduce $I_j(N_{i+j+1}) = x - 2$ and $L_j(N_{i+j}) = x$. Invoking Lemma 2, we deduce that $I_{j+1}(N_{i+j}) = I_{j+1}(N_{i+j+1}) \geq x - 1$.

Since $I_j(N_{i+j-1}) \leq x$ by the induction assumption, there is no violation between N_{i+j-1} and N_{i+j}. There is obviously no violation between N_{i+j} and N_{i+j+1}, since both loads are in the same interval. The only possible violation might be between N_{i+j+1} and N_{i+j+2}, which is permitted under the induction assumption. It is also clear that both $I_{j+1}(N_{i+j})$ and $I_{j+1}(N_{i+j+1})$ are at most x, thus completing the induction step.

The above inductive proof, combined with the fact that the procedure terminates when the last node is reached, shows that there are no NBRBALANCE violations when ADJUSTLOAD terminates. □

Lemma 4. *Consider a state of the system where both* NBRBALANCE *and* GLOBALBALANCE *invariants hold. If a tuple insert now causes a violation of* GLOBALBALANCE, *the consequent execution of* ADJUSTLOAD *will ensure both* GLOBALBALANCE *and* NBRBALANCE.

Proof. Consider the insert of a tuple at node N_i. By definition, no violation of GLOBALBALANCE arises unless the load of N_i crosses a threshold. Suppose $L(N_i)$ crosses a threshold, say T_x. There may then be (a) a violation of NBRBALANCE between N_i and either or both of its neighbors, (b) a violation of GLOBALBALANCE between N_i and some other non-neighbor node N_j.

In case (a), node N_i performs a NBRADJUST, which drops its load to at most T_x (as shown in Lemma 3), and therefore eliminates any GLOBALBALANCE violation involving N_i. As Lemma 3 shows, all other nodes involved in NBRADJUSTs only experience an increase in load, and have a load no more than T_x, thus ruling out any other GLOBALBALANCE violation.

If case (a) does not arise, then there is no NBRBALANCE violation, but there may still be a GLOBALBALANCE violation involving N_i. In this case, Procedure ADJUSTLOAD performs REORDER using the least-loaded node in the system, say N_j. W.l.o.g., assume that N_j transfers its load to N_{j+1} before splitting the load of N_i. We first show that there are no GLOBALBALANCE violations after this REORDER operation.

Let $L_0(.)$ refer to node loads before REORDER, and $L_1(.)$ the loads after REORDER. Then, $L_1(N_i) = \lceil L_0(N_i)/2 \rceil$, $L_1(N_j) = \lfloor L_0(N_j)/2 \rfloor$, and $L_1(N_{j+1}) = L_0(N_j) + L_0(N_{j+1})$. By Lemma 1 properties (c),(d) and (e), we deduce that $I(N_{j+1} \leq x$, $I(N_i) \in [x-1, x]$ and $I(N_j) \in [x-1, x]$ respectively, thus eliminating all GLOBALBALANCE violations.

We also observe that there are no NBRBALANCE violations between N_{i-1} and N_i or between N_j and N_{i+1}, since $I(N_{i-1}) = I(N_{i+1}) = x$. (If they were greater than x, there must have been a GLOBALBALANCE violation involving them, and if they were less than x, N_i would have performed a NBRADJUST.)

However, it is possible to have NBRBALANCE violations around node $A = N_{j+1}$ (with both node N_{j+2} and N_{j-1}). Let us now re-index the nodes after the REORDER operation, and say node A becomes N_k after re-indexing. We are now left with a state where there may be NBRBALANCE violations at $(k-1, k)$ and $(k, k+1)$. In addition, for any node N, we know $I(N) \in [x-2, x]$ by the GLOBALBALANCE property.

If there is a NBRBALANCE violation, ADJUSTLOAD executes at node N_k. Let there be a violation between nodes N_k and N_{k+1}. In this case, NBRADJUST occurs between N_k and N_{k+1}, followed by a recursive ADJUSTLOAD on N_{k+1}. As shown in Lemma 3, this results in the elimination of all NBRBALANCE violations involving N_{k+r} for any $r > 0$.

However, it is still possible for a NBRBALANCE violation to exist between N_{k-1} and N_k, if $I(N_k) = x$ and $I(N_{k-1}) = x-2$. In this case, ADJUSTLOAD continues to execute at node N_k, and performs a NBRADJUST with N_{k-1}, followed by a recursive call to ADJUSTLOAD on N_{k-1}. Again by Lemma 3, this results in the elimination of all NBRBALANCE violations involving N_{k-r} for any $r > 0$.

The final step is to show that N_k and N_{k+1} do not cause a NBRBALANCE violation, even after the NBRADJUST between N_{k-1} and N_k. This is true, since the final value of both $I(N_k)$ and $I(N_{k+1})$ is at least $x-1$ by Lemma 2. □

Proof of Theorem 3. We prove that the invariants hold by induction on the length l of the insert/delete sequence. The invariants clearly hold when $l = 0$ since all nodes contain one sentinel value.

Assume that the invariants hold for $l = r$. Let the $(r+1)^{st}$ operation be an insert. If this insert does not violate any invariants, we are done. If not, either NBRBALANCE or GLOBALBALANCE is violated. We have shown that all such violations are fixed by ADJUSTLOAD in Lemmas 3 and 4 respectively.

If the $(r+1)^{st}$ operation is a delete, it is straightforward to show that all violations are again fixed, by a proof similar to that of Lemmas 3 and 4. We have thus proved that the invariants hold after any sequence of inserts and deletes. □

Corollary 3.1. *For the Threshold algorithm, the imbalance ratio σ is δ^3 after any sequence of inserts and deletes.* ■

Proof. At any point in time, all loads $L(N_i)$ lie in the interval $(T_{r-3}, T_r]$, for some value of r. Since $T_r \leq T_{r-3} \times \delta^3 + 1$, the imbalance ratio is δ^3. □

Corollary 3.2. *For the Fibbing algorithm, the imbalance ratio σ is ϕ^3 after any sequence of inserts and deletes.* ■

Theorem 4. *For the Threshold (and Fibbing) algorithm, the amortized cost of both inserts and deletes is constant.*

Proof. We bound the amortized costs of insert and delete by using the potential method. Let $\bar{L}$ denote the current average load. Consider the potential function $\Phi = c(\sum_{i=1}^{n} L(N_i)^2)/\bar{L}$, where c is a constant to be specified later. We will show the following: (a) the cost of NBRADJUST is bounded by the drop in potential accompanying it, (b) the cost of REORDER is bounded by the drop in potential accompanying it, and (c) the gain in potential after a tuple insert or delete, before any rebalancing actions, is bounded by a constant. The above three statements together imply that the amortized costs of tuple insert and delete are constant.

NBRADJUST: Recall that a NBRADJUST operation occurs between two nodes whose load differs by at least a factor δ. Let the loads of the two nodes involved be x and y. The drop in potential $\Delta\Phi$ from NBRADJUST is $c(x^2 + y^2 - (x+y)^2/2)/\bar{L} = c(x-y)^2/2\bar{L}$. By Lemma 1(f), $x - y > (\delta - 1)y$, and y is at least $\bar{L}/\delta^3$. Therefore, $\Delta\Phi > c'(x-y)$ for some constant c'. Since the number of tuples moved is at most $(x-y)/2$, the drop in potential pays for the data movement by choosing the constant c to be sufficiently large ($> \delta^3/(\delta-1)$).

REORDER: Let a REORDER operation involve a node with load x, and a pair of neighbor nodes with loads y and z, with $y \leq z$. We then have $\delta^2 y \leq x$ (for the REORDER operation to be triggered), and $\delta z \leq x$ (by NBRBALANCE between the neighbors).

The drop in potential from REORDER is given by:

$$\begin{aligned}\Delta\Phi &= c(x^2 + y^2 + z^2 - 2(x/2)^2 - (y+z)^2)/\bar{L} \\ &= c(x^2/2 - 2yz)/\bar{L} \geq c(x^2/2 - 2x^2/\delta^3)/\bar{L} \\ &\geq c'x(1 - 4/\delta^3)\end{aligned}$$

Note that $1 - 4/\delta^3$ is greater than zero for $\delta > \sqrt[3]{4} \simeq 1.587$. The data movement cost of REORDER is $\lfloor x/2 \rfloor + y < x$. Therefore, for a suitable choice of constant c ($> 2\delta^3/(\delta^3 - 4)$), the drop in potential pays for data movement.

Tuple Insert: The gain in potential, $\Delta\Phi$, from an insert at node N_i and before any rebalancing, is at most $c((L(N_i)+1)^2 - L(N_i)^2)/\bar{L}$, where $\bar{L}$ refers to the average load before the latest insert. Therefore, $\Delta\Phi \leq c(2L(N_i)+1)/\bar{L} \leq c(2\delta^3+3)$, since $L(N_i) \leq \delta^3\bar{L}+1$ and $\bar{L} \geq 1$. Therefore, the amortized cost of an insert is constant.

Tuple Delete: When a tuple is deleted, there may be a gain in potential due to a slight reduction in $\bar{L}$. Since $\bar{L}$ drops in value $1/n$ from a delete, the maximum gain in potential $\Delta\Phi = \frac{c(\sum L(N_i)^2)(1/n)}{\bar{L}(\bar{L}-1/n)}$. Using the facts $L(N_i) \leq \delta^3\bar{L}+1$, $\bar{L} \geq 1$, and $n \geq 2$, we can see that $\Delta\Phi \leq c(5\delta^3+3)$. Therefore, the amortized cost of a delete is constant. $\square$

We observe that the bounds on these amortized costs are quite large. When $\delta = \phi$, the cost of an insert $\simeq 412$, and the cost of a delete $\simeq 868$. We believe this to be a consequence of weak analysis stemming from our choice of potential function. We show experimentally in Section 5 that these constants are actually very close to 1. We also present variations of our algorithm next that are amenable to tighter analysis.

3.4 Discussion

Improving σ further It is possible to improve σ to arbitrarily small values larger than 1, by generalizing the Threshold algorithm, and maintaining balance over larger sets of consecutive nodes, rather than just pairs of neighbors. We do not detail this generalization in this work.

The Doubling Algorithm with Hysteresis It is possible to define a variant of the Doubling Algorithm which provides a weaker imbalance ratio ($\sigma = 32$) but has provably stronger bounds on the insertion and deletion costs. The idea is to use *hysteresis*, and require a node to lose at least half its data before it triggers load balancing for tuple deletions. We can show that this variant guarantees insertion cost of 4 and deletion cost of 29.

A Randomized Variant So far, all our algorithms attempt to find the least-loaded node (or the most-loaded node) in order to initiate the REORDER operation. In fact, the theorems we have stated hold even for a slightly weaker condition: If there are multiple nodes that violate the GLOBALBALANCE condition with respect to a particular node N_i executing ADJUSTLOAD, it suffices for N_i to attempt the REORDER operation with *any* of these nodes.

Such a weakening suggests an interesting randomization which avoids trying to find the least-loaded node altogether. Node N_i simply samples a set of ρ nodes at random. If one of them violates GLOBALBALANCE, N_i performs the REORDER operation using this node; otherwise, N_i simply does nothing.

If there are no data deletes, this randomized algorithm guarantees that the maximum load is at most a constant factor times the *average load* with high probability, so long as the number of nodes sampled ρ is $\Theta(\log n)$. In the presence of tuple deletes, we offer a different guarantee: If a node N_i specifies a *peak threshold* C that N_i does not want its load to exceed, the load of N_i does not exceed C with high probability, unless the average load in the system $\bar{L}$ is within a constant factor of C. It is also possible to provide guarantees on the imbalance ratio in the presence of deletes, but more caveats need to be added to the algorithm. [4]

Concurrency Control and Parallelism Until now, we have assumed that tuple inserts (and deletes) happen in sequence, and that a load-balancing step completes before the next insert. Our algorithms generalize naturally to (a) deal with parallel inserts that may happen before a load-balancing step completes, and (b) allows multiple load-balancing steps to execute in parallel.

While we do not discuss either of the above issues in detail here, we make the following observations and claims. First, the load-balancing step can be broken into multiple, smaller, atomic actions that require only simple block-level locking. Second, tuple inserts and deletes may be given higher priority, and allowed to execute even before the full load-balancing step for a previous insert/delete completes. Third, multiple load-balancing steps can execute in parallel and require very simple serialization mechanisms to ensure correctness. Finally, we note that it is possible to formalize a model of concurrency in which we can characterize the imbalance ratio under parallel insertions and deletions.

4 A P2P Network for Range Queries

There has been recent interest in developing P2P networks that can support ad-hoc queries over key ranges [3, 4, 20]. A solution is to use range partitioning of data across the peer nodes. If the data and query distributions are uniform, nodes will have equal loads. However, if the data and/or execution is skewed, the network will develop hot-spots with high query traffic for a few nodes. Load balancing thus becomes a critical requirement in such a system. The P2P environment imposes three significant challenges for developing a load-balanced, range-partitioned system:

Scale The size of the system may be extremely large, upto tens or hundreds of thousands of nodes. Our load-balancing algorithm deals well with scale, since its data-movement cost is *constant* and independent of the number of nodes.

Dynamism The lifetime of nodes is short (a few hours) and arbitrary (whims of the node owner). Our load-balancing algorithms need to efficiently handle dynamic arrival and departure of nodes while ensuring good load balance across the existing nodes in the system. We discuss this adaptation in Section 4.1.

Decentralization P2P systems do not have a central site that collects statistics and routes queries/inserts/deletes. We need distributed routing and data structures to enable both routing of queries/inserts/deletes, as well as to find additional load information for load balancing. Maintain-

[4]Karger and Ruhl [14] offer a randomized algorithm that provides such guarantees, but require *each* node to perform such random sampling and rebalancing, whether or not any inserts or deletes are directed to that node.

ing such data structures also imposes additional costs on the system, as we discuss in Section 4.2.

4.1 Handling Dynamism in the Network

Node Arrival Upon arrival, a new node N finds the most-loaded node N_h in the network. It then splits the range of N_h to take over half the load of N_h, using the NBRADJUST operation. After this split, there may be NBRBALANCE violations between two pairs of neighbors: (N_{h-1}, N_h) and (N, N_{h+1}). In response, ADJUSTLOAD is executed, first at node N_h and then at node N. It is easy to show (as in Lemma 3) that the resulting sequence of NBRADJUST operations repair all NBRBALANCE violations.

Node Departure While in the network, each node manages data for a particular range. When the node departs, the data it stored becomes unavailable to the rest of the peers. P2P networks reconcile this data loss in two ways: (a) Maintain replicas of each range across multiple nodes. A common scheme for replication is to ensure that the partition of node N_i is replicated at the preceding r nodes (with N_n preceding N_1), for a system-specified constant r [8, 18]. (b) Do nothing and let the "owners" of the data deal with its availability. The owners will frequently poll the data to detect its loss and re-insert the data into the network.

First, consider the simpler data-is-lost case (b). Here, when a node N_i departs, the range boundaries between N_{i-1} and N_{i+1} must be modified. There could be a NBRBALANCE violation between the new neighbors (N_{i-1}, N_{i+1}) which can be fixed by N_{i-1} executing ADJUSTLOAD. As shown in Lemma 3, this is sufficient to restore the system invariants.

Now consider the data-is-replicated case (a). Here, when a node N_i departs the network, its preceding node N_{i-1} assumes management of N_i's partition. The node N_{i-1} already has N_i's data replicated locally. We can consider the new state as being logically equivalent to a node departure in the data-is-lost case (b), followed by a subsequent insertion of the "lost" tuples by N_{i-1}. The load-balancing algorithm is initiated whenever such insertion makes a node's load cross a threshold.

The Costs of Node Arrival and Departure The data-movement cost of a node arrival and departure is straightforward to analyze. When a new node arrives, it receives half the load of the largest node, thus requiring $\Theta(\bar{L})$ data movements, where $\bar{L}$ is the average load per node after node arrival (since all node loads are within a constant factor of each other). In addition, the average load per node decreases, leading to an increase in potential of $\Theta(\bar{L})$. Thus, the amortized cost of node insertion is still $\Theta(\bar{L})$. Note that this cost is asymptotically optimal, since it is impossible to achieve a constant imbalance ratio without the new node receiving at least $\Theta(\bar{L})$ tuples.

In the data-is-lost case, the data-movement cost of a node departure is 0, since a node departure only raises the average load, resulting in a drop in potential. All subsequent NBRADJUST operations pay for themselves, as discussed earlier. In the data-is-replicated case, the data-movement cost of a node departure is equal to the cost of "re-insertion" of the "lost" data; since the amortized cost of each insert is constant, and we re-insert only $O(\bar{L})$ tuples, the amortized cost of node departure is $O(\bar{L})$. This is again asymptotically optimal.

Note that in the replicated case, both arrival and departure of nodes requires re-creation of lost replicas, or migration of existing ones. Similarly, tuple inserts and deletes also have to be duplicated at the replica nodes. We presume that such replication is performed in the background. Observe that such replica maintenance inflates the costs of all operations only by a constant factor, if the number of replicas r is a constant.

4.2 Dealing with Decentralization

So far, we have assumed the existence of a central site that (a) maintains the global partitioning information to direct queries appropriately, and (b) maintains global load information for the load-balancing algorithm to exploit. Our next step is to devise decentralized data structures to perform both the above functions. We first describe a known data structure for efficient range queries, before discussing how to maintain load information.

Cost Model In the centralized setting of Section 2, we considered only data-movement cost, and ignored the cost of maintaining partition and load information. In the P2P setting, the lack of a central site means that we can no longer ignore this latter cost. Therefore, each operation (query, tuple insert, tuple delete, node insert, node delete) is now associated with two different costs: (a) the data-movement cost, which is measured just as earlier, and (b) communication cost, defined to be the number of *messages* that need to be exchanged between nodes (to maintain and probe the data structure) in order to perform the operation (and its corresponding load-balancing actions). Note that each message is between a pair of nodes, i.e., communication is point-to-point and broadcast is not free. (We will ignore the cost of returning query answers to the querying node.)

A Data Structure for Range Queries Our first task is the following: Any node receiving a query/insert/delete should be able to efficiently forward the operation to the appropriate node(s). One solution is to replicate the partition information across all nodes. A node, upon receiving a query, can simply look up this information and forward the request directly to the appropriate node(s). However, every node insertion/deletion, or partition change, needs to be broadcast to all nodes, which is very inefficient. On the other extreme, nodes could be organized in a linked list, ordered by the partitions they manage. Updating the data structure on partition changes or node arrival/departure is efficient, but queries now have to traverse the entire linked list to reach the appropriate node.

A compromise between these two costs may be achieved using a data structure known as the skip graph [6, 13]. Skip graphs are essentially circular linked lists, but each node

Operation	Messages (w.h.p)	Data Movement
Tuple Insert	$O(\log n)$	$O(1)$
Tuple Delete	$O(\log n)$	$O(1)$
Node Arrival	$O(\log n)$	$O(\bar{L})$
Node Departure	$O(\log n)$	0 or $O(\bar{L})$
Lookup Query	$O(\log n)$	0
Range Query	$O(\log n + fn)$	0

Table 1: *Cost of operations supported by the P2P network. The parameter f denotes the selectivity of the range query. The data-movement cost of Node Departure depends on the model used for data loss.*

also maintains roughly $\log n$ *skip pointers*, to enable faster list traversal. Skip pointers are randomized, and routing between any two nodes requires only $O(\log n)$ messages with high probability. Consequently, a query can be forwarded from any node to the first node in the query's range, say N_1, using $O(\log n)$ messages. If the query range is large and spans q nodes, the query is simply forwarded along on the linked list to the q successors of N_1. When a node arrives or departs, only $O(\log n)$ messages are required to update the data structure. Partition changes due to NBRADJUST do not require any messages at all.

Maintaining Load Information Our algorithm requires that each node be able to find (a) the load of its neighbors, and (b) the most and least-loaded node in the system. Dealing with problem (a) is easy: a node already has links to its two neighbors in the skip graph, thus requiring just one message each to find their loads.

To deal with problem (b), we simply build a second, separate skip graph *on the node loads*. In other words, nodes are arranged in a sequence sorted by their current load (with ties broken arbitrarily), and a skip graph is constructed on this sequence. As node loads change, the sequence may have to be updated, but it will turn out that such updates are not expensive. As discussed earlier, this data structure enables discovery of the most and least-loaded node with just $O(\log n)$ messages, while also enabling efficient updates to the data structure.

As mentioned in Section 3.4, it is not necessary for a node to always find the most or least-loaded node, so long as it locates any node that violates GLOBALBALANCE. This property allows us to terminate searches on the skip graph even before locating the most or least-loaded node. The early termination mitigates "hot spots" created when multiple nodes simultaneously seek the most-loaded node.

The P2P Structure and its Costs: A Summary We summarize all the operations and their costs supported by the P2P system in Table 4.2. The bounds on the message costs of operations follow directly from our discussion of skip graphs above. We note that the data-movement costs are amortized, while the message costs hold with high probability. We observe that the above costs are asymptotically identical to the costs in DHTs, except for range queries where our structure is more efficient.

5 Experimental Evaluation

In this section, we present results from our simulation of the Threshold algorithm on networks ranging in size from $n = 2^4$ to 2^{14}. We compare the performance of our algorithm against periodic reorganization. We also evaluate our adaptations of the algorithm on a P2P network. Our simulations show the following results:

A The Threshold Algorithm achieves the desired imbalance ratio for a range of σ values on various workloads.
B The amortized cost of load balancing is very small, decreases with increasing σ, and is much lower than the cost of periodic reorganization.
C The P2P variant achieves the desired imbalance ratio at a small cost, and scales gracefully with increasing dynamism in the network.
D The Randomized variant provides good imbalance ratios even with a small number of samples.

5.1 Simulation Model

In the parallel database setting, the simulation is designed to study load balancing as the relation evolves over time. The system is studied under three phases: (a) Growing, (b) Steady, and (c) Shrinking. At the start, all n nodes in the system are empty ("cold start"). In the Growing phase, data is loaded, one tuple at a time, using a sequence of $D = 10^6$ insert operations. In the Steady phase, inserts and deletes alternate for a total of D operations. In the following Shrinking phase, data is removed from the system, one tuple at a time, using a sequence of D delete operations.

The workload, i.e., the sequence of insertions and deletions, is set to be one of the following:

A ZIPFIAN models a static data distribution. Each tuple inserted in the Growing and Steady phases has an attribute A drawn from a Zipfian distribution (with parameter 1.0) with values in the range $[1, 10000]$. (Since our range partitioning operates on a relational key, we create a unique attribute B for each tuple, and use the sequence $\langle A, B\rangle$ as the ordering key for range partitioning.) Tuple deletion during the Steady and Shrinking phases removes one of the existing tuples uniformly at random.
B HOTSPOT models a skewed workload in which all inserts and deletes are directed to a single pre-selected ("hot") node.
C SHEARSTRESS models a dynamic workload in which an "adversary" inspects the load of nodes after each insert or delete of a tuple. The adversary then constructs the following insert (or delete) such that it is routed to the current most-loaded (resp. least-loaded) node.

We study the effects of network dynamism on load-balancing for a P2P network under a similar evolution model. The network starts with an initial $n = 16$ nodes, into which D tuples are inserted one by one. In the Growing phase, nodes arrive one by one and join the network, until $n = 1024$. In the following Shrinking phase, nodes depart at random until the network shrinks to $n = 16$. We use data replication to ensure that tuples are not lost on

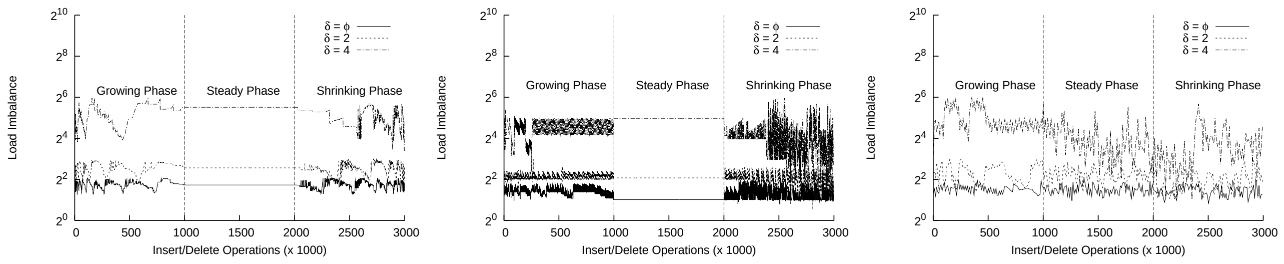

Figure 3: Imbalance ratio for (a) ZIPFIAN, (b) HOTSPOT, and (c) SHEARSTRESS when $n = 256$.

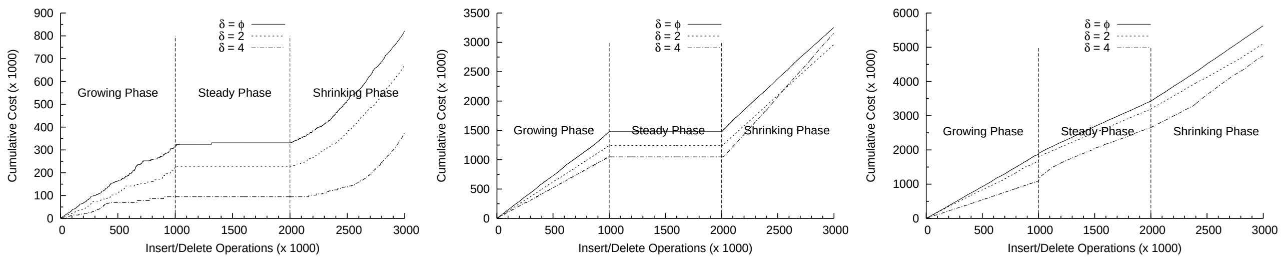

Figure 4: Data movement costs on the (a) ZIPFIAN, (b) HOTSPOT, and (c) SHEARSTRESS workload when $n = 256$.

node departures. No tuples are inserted or deleted during the Growing and Shrinking phases: our goal is to isolate the costs of node arrival/departure on load balancing.

5.2 Imbalance Ratio

We start with an evaluation of imbalance ratios ensured by the Threshold Algorithm for various workloads. For the experiments, we measure the imbalance ratio at any instant as the ratio of the largest and smallest loads at that instant (with all loads being at least 1). Figure 3 shows the imbalance ratio (Y-axis) against the number of insert and delete operations (X-axis) during a run on (a) ZIPFIAN, (b) HOTSPOT, and (c) SHEARSTRESS workloads with 256 nodes. The curves are drawn for $\delta = \phi$ (Fibbing Algorithm), $\delta = 2$ (Doubling Algorithm) and $\delta = 4$.

We observe that Threshold Algorithm ensures that imbalance ratio is always less than δ^3 for all δ. Each spike in the curve corresponds to an ADJUSTLOAD step. As δ increases, the jitter introduced by the spikes gets larger and larger; this is because the algorithm allows the imbalance ratio to worsen by a *constant factor*, roughly δ, before load balancing occurs. The curves are smooth in the Steady phase for ZIPFIAN and HOTSPOT. For the former, the range partitioning "adapts" to the data distribution, ensuring that inserts and deletes are randomly sprinkled across nodes; for the latter successive inserts and deletes occurring at the same node cancel out. However, the adversary in SHEARSTRESS picks its inserts and deletes carefully to cause imbalance, leading to continuous variation in σ.

5.3 Data Movement Cost

We next study the data movement cost incurred by the Threshold Algorithm for ensuring balance in the runs discussed above. Figure 4 plots the cumulative number of tuples moved by the algorithm (Y-axis) against the number of insert and delete operations (X-axis) during a run.

We observe that costs for different δ are roughly the same (within 20% of each other) for the HOTSPOT and SHEARSTRESS workloads. Intuitively, this is because keeping the system tightly balanced causes a larger number of rebalancing operations, but each operation has lower cost due to the tight balance. We also observe that there is no data movement in the Steady phase for ZIPFIAN, indicating that the system has "adapted" to the data distribution. For the other two phases, the curves are linear confirming that the amortized cost per operation is constant, and independent of the amount of data in the system. The constants involved are also very small, with the cost per insert/delete, even in the worst phase, being roughly 0.3, 1.5 and 2 for the three workloads.

To put the above performance in perspective, we compared the data movement costs of the Fibbing Algorithm against those incurred by a periodic reorganization strategy that ensures the same imbalance ratio $\sigma = 4.2$ bound as follows: the central site continuously observes σ, and whenever $\sigma > 4.2$, a reorganization is triggered to create a perfectly balanced set of nodes. The reorganization identifies a balanced placement of tuples across nodes, and then moves each tuple *at most once* by sending it directly to its final destination node. (Thus, it is more efficient than using only NBRADJUST operations.) Figure 5(a) plots the cumulative data movement costs on a logarithmic scale (Y-axis) against the number of operations (X-axis) in a run on 256 nodes. We observe that the periodic reorganization performs nearly 10 times *worse* for ZIPFIAN and upto 50 times worse for others. The reasons are two-fold: (a) its non-online nature allows the skew to grow requiring an expensive clean-up, and (b) its perfect balancing causes more data movement than essential to obtain the desired bounds.

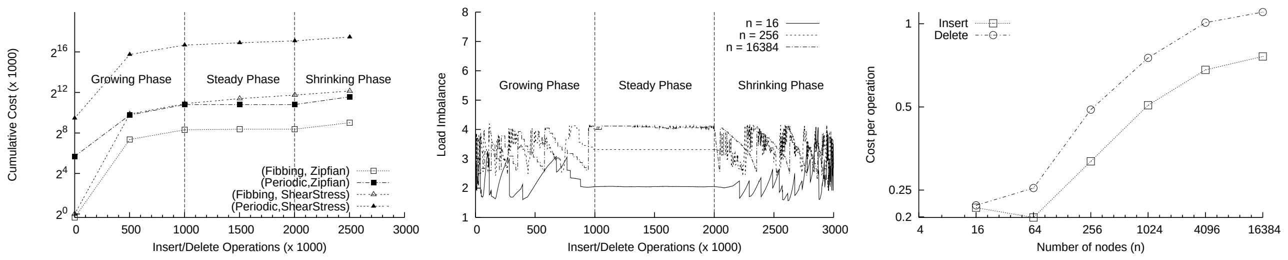

Figure 5: (a) Data movement costs for Fibbing Algorithm compared to periodic reorganization (b) Effect of n on imbalance ratios (c) Effect of n on data movement cost.

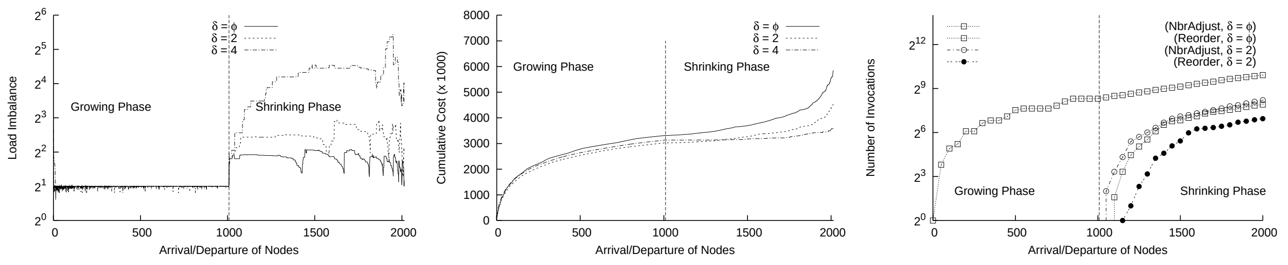

Figure 6: Performance in a P2P system: (a) imbalance ratios, (b) data movement costs, and (c) number of NBRADJUST and REORDER invocations for the Threshold Algorithm on the ZIPFIAN workload.

5.4 The Effects of Scaling

We next study the effects of scaling in the number of nodes n on the performance of the Threshold Algorithm. Figures 5(b) and 5(c) plot load imbalance and data movement cost for the Fibbing Algorithm against a run on the ZIPFIAN workload. The network size n is varied from 16 to 16384. We observe in Figure 5(b) that the Fibbing Algorithm continues to ensure the $\sigma = \phi^3$ bound. However, as the same number of tuples are shared across more nodes, the load variance across nodes increases, leading to an increase in the imbalance ratio.

Figure 5(c) plots the data movement cost per operation (Y-axis) against the size of the network n (X-axis). Both axes are plotted on a logarithmic scale. The bottom curve plots the data movement cost per insert observed during the Growing Phase; the top curve plots the costs per delete observed during the Shrinking Phase. We observe that costs of both insert and delete operations increases with increasing n. As n increases, the load per node is smaller, making it easier to make the system unbalanced with a smaller number of inserts and deletes. Thus more load balancing steps are needed, leading to a higher cost. We also observe that the cost per operation is quite small as the curves taper off towards a value close to 1.

We had shown in Section 3.3 that the cost per insert or delete is a *constant*. The figures here show a dependence of cost per operation on n. How can this apparent contradiction be explained? The experiments presented here evaluate the cost for a *fixed* workload on various n values. On the other hand, the analysis established bounds on the *worst-case* costs against all workloads.

5.5 Performance in a P2P Setting

Figure 6 shows the performance of the Threshold Algorithm adapted to a P2P system. Figures 6(a) and 6(b) plot the imbalance ratio and data-movement cost against the number of node arrivals and departures. We observe in Figure 6(a) that the system remains well-balanced through the Growing phase, because the arriving node always splits the most-loaded node. The imbalance ratio is roughly two, since the most-loaded node splits its load in half on a node arrival. On the other hand, nodes depart at random during the Shrinking phase, which leads to changes in the imbalance ratio. However, the guarantees of δ^3 are ensured.

From Figure 6(b), we see that the *incremental* data-movement cost per arrival/departure (i.e., the slope of the curve) decreases with node arrivals in the Growing phase, and increases with node departures in the Shrinking phase. This is not surprising, since the cost is proportional to the average load in the system which, in turn, is inversely proportional to the number of nodes.

NbrAdjust vs. Reorder In a P2P system, the NBRADJUST operation may turn out to be more efficient than REORDER, for two reasons: (a) REORDER requires the reordered node to drop its old links and set up new ones. (b) A NBRADJUST may not require immediate transfer of data to balance load, when data is replicated at neighbors; only replicas need to be updated, which can be done more lazily.

In the light of the above observations, we observed the number of invocations of the two operations to see which is used more often. Figure 6(c) shows the number of invocations of NBRADJUST and REORDER for different values of δ, as nodes are inserted and deleted. Not surprisingly, the number of invocations of both operations decreases as

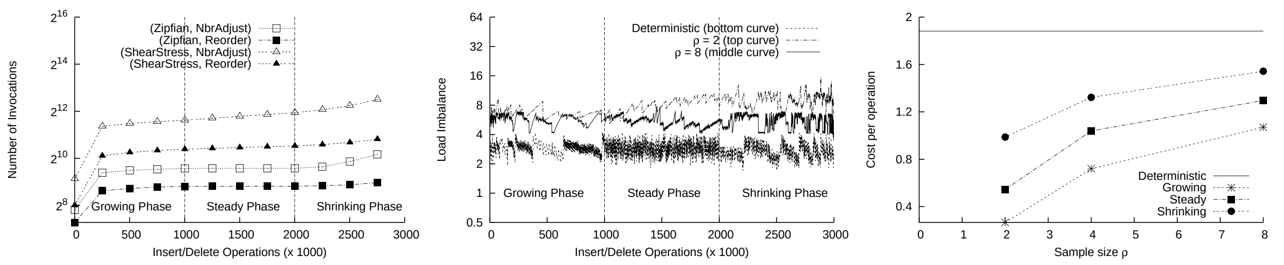

Figure 7: The Fibbing algorithm with $n = 256$. (a) The number of invocations of NBRADJUST and REORDER (b) The effect of randomization on imbalance ratio (c) The effect of randomization on data movement costs, for the SHEARSTRESS workload.

δ increases. We see that the number of NBRADJUST invocations is at least 4 times that of REORDER, which is reassuring given that REORDERs are more expensive.

Figure 7(a) shows the number of invocations of the two operations by the Fibbing Algorithm, on a *fixed* set of 256 nodes, as tuples are inserted and deleted from the three workloads. We observe that there are twice as many invocations of NBRADJUST, as compared to REORDER, which is again useful in the P2P context.

5.6 The Effects of Randomization

As discussed earlier, the REORDER operation requires global statistics and involves the least/most-loaded node in the load-balancing step. We defined a randomized variant of Threshold Algorithm in Section 3.4 where REORDER would sample ρ nodes at random and pick the least/most-loaded node from the *sample*. Figures 7(b) and 7(c) plot the effects of such randomization on the Fibbing Algorithm for runs of the SHEARSTRESS workload as the sample size ρ is varied. We observe in Figure 7(b) that the imbalance ratio (Y-axis) degrades beyond the original value of ϕ^3. However, even the use of $\rho = 2$ samples provides good imbalance ratios, and increasing ρ improves σ further. Correspondingly, the use of random sampling results in less data movement compared to the deterministic case, for all three phases, as shown in Figure 7(c).

6 Related Work

Parallel Databases partition relations across multiple disks, using either range or hash partitioning [7, 9, 23]. Research in physical design of parallel databases can be classified into four categories: (a) Performing workload-driven tuning of storage for static relations, e.g. [12, 16, 21]; (b) Designing disk-based data structures for fast bulk insert/delete of tuples, e.g. [11, 15]; (c) Enabling efficient data migration for load-balancing while allowing concurrent relation updates and queries, e.g. [24]; and (d) Balancing query load across disks by transferring a partition from one disk to another, e.g. [19].

Work in category (a) is focused on performing workload-driven tuning of physical design, but usually does not consider a dynamic evolution of the design with relation updates. Work in category (b) is complementary to ours, in that they show *how* to efficiently update *local* disk structures when tuples move from one partition to another, while our focus is in understanding *what* tuples to move. Research in category (c) is also complementary to our work, as it helps deal with issues of concurrency control when performing online repartitioning. Finally, work in category (d) attempts to modify the *allocation* of partitions to disks, rather than change the partitions themselves. We believe such solutions could be used in combination with ours to achieve balance for dynamic query loads, but are not sufficient in themselves to guarantee storage balance for range-partitioned data.

Range Queries in P2P Networks Recently, P2P networks supporting range queries have been proposed, that offer either storage balance or efficient queries, but not both. Ratnasamy et. al. [20] assure storage balance but at the price of data-dependent query cost and data fragmentation. Gupta et. al. [3] provide approximate answers, and do not offer any guarantees for an arbitrary range query. Others [4, 6, 13] offer exact and efficient queries, but do not offer load balance across nodes.

Aberer et al. [1, 2] develop a P2P network called P-Grid that can support efficient range queries. All nodes in this system are assumed to have a fixed capacity. The system *heuristically* replicates content to fill all the nodes' capacity. However, there is no formal characterization of either the imbalance ratio guaranteed, or the data-movement cost incurred in achieving load balance. In addition, the algorithm requires every node to periodically contact other nodes in the system to gather statistics about node loads.

In a concurrent work, Karger and Ruhl [14] provide an alternative solution to the storage balance problem. The scheme is a randomized algorithm that offers a high-probability bound on the imbalance ratio, and is analyzed under a dynamic, but non-adversarial, setting. However, the best achievable bound on imbalance ratio using this algorithm appears to be more than 128, which is much higher than the load imbalance bounds we guarantee.

Routing in P2P Networks Most DHT interconnection networks (e.g., Pastry [18], Chord [22]) require randomly chosen (uniformly-spaced) partition boundaries to guarantee efficient routing in $O(\log n)$ messages. A load-balanced, range-partitioned network will not have such equi-spaced boundaries, thus rendering such DHT structures unusable.

Aberer [1] presents an elegant variant of Pastry which does guarantee $O(\log n)$ routing even with arbitrary partition boundaries. However, there is a risk that node in-degrees can become skewed, resulting in a skewed message traffic distribution. Moreover, a change of partition boundaries between neighbors in load-balancing will necessitate a change in the network link structure. Our P2P network utilizes skip graphs which overcomes the above limitations. Finally, we have developed a Chord-like variant of skip graphs that we could use instead of skip graphs in our solution.

7 Conclusions

Horizontal range-partitioning is commonly employed in shared-nothing parallel databases. Load balancing is necessary in such scenarios to eliminate skew. We presented asymptotically optimal online load-balancing algorithms that guarantee a constant imbalance ratio. The data-movement cost per tuple insert or delete is constant, and was shown to be close to 1 in experiments.

Our solutions were motivated by a new application domain for range-partitioning: peer-to-peer (P2P) systems. We showed how to adapt our algorithms to dynamic P2P environments, and architected a new P2P system that can support efficient range queries.

Although our algorithms were presented in the context of balancing storage load, they directly generalize to balancing dynamic query load too. We believe our work is useful even when data is hash-partitioned, since hash-partitioning does not guarantee storage balance by itself unless the hash attribute is a relational key.

References

[1] K. Aberer. Scalable data access in p2p systems using unbalanced search trees. In *Proc. WDAS*, 2002.

[2] K. Aberer, A. Datta, and M. Hauswirth. The quest for balancing peer load in structured peer-to-peer systems. Technical Report IC/2003/32, EPFL, Switzerland, 2003.

[3] A.Gupta, D.Agrawal, and A. Abbadi. Approximate range selection queries in peer-to-peer systems. In *Proc. CIDR*, 2003.

[4] A. Andrzejak and Z. Xu. Scalable, effi cient range queries for grid information services. In *Proc. P2P*, 2002.

[5] A.Silberschatz, H.F.Korth, and S.Sudarshan. *"Database System Concepts"*, chapter 17. McGraw-Hill, 1997.

[6] J. Aspnes and G. Shah. Skip graphs. In *Proc. SODA*, 2003.

[7] G. Copeland, W. Alexander, E. Boughter, and T. Keller. Data placement in Bubba. In *Proc. SIGMOD*, 1988.

[8] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. SOSP*, 2001.

[9] D. DeWitt, R. H. Gerber, G. Graefe, M. L. Heytens, K. B. Kumar, and M. Muralikrishna. Gamma -a high performance dataflow database. In *Proc. VLDB*, 1986.

[10] D. DeWitt and J. Gray. Parallel database systems: The future of high performance database processing. *Communications of the ACM*, 36(6), 1992.

[11] H. Feelifl, M. Kitsuregawa, and B. C. Ooi. A fast convergence technique for online heat-balancing of btree indexed database over shared-nothing parallel systems. In *Proc. DEXA*, 2000.

[12] S. Ghandeharizadeh and D. J. DeWitt. A performance analysis of alternative multi-attribute declustering strategies. In *Proc. SIGMOD*, 1992.

[13] N. J. A. Harvey, M. Jones, S. Saroiu, M. Theimer, and A. Wolman. Skipnet: A scalable overlay network with practical locality properties. In *Proc. USITS*, 2003.

[14] D. R. Karger and M. Ruhl. Simple effi cient load-balancing algorithms for peer-to-peer systems. In *Proc. IPTPS*, 2004.

[15] M. L. Lee, M. Kitsuregawa, B. C. Ooi, K.-L. Tan, and A. Mondal. Towards self-tuning data placement in parallel database systems. In *Proc. SIGMOD*, 2000.

[16] J. Rao, C. Zhang, N. Megiddo, and G. Lohman. Automating physical database design in a parallel database. In *Proc. SIGMOD*, 2002.

[17] S. Ratnasamy, P. Francis, M. Handley, and R. M. Karp. A scalable Content-Addressable Network. In *Proc. SIGCOMM*, 2001.

[18] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, distributed object location, and routing for large-scale peer-to-peer systems. In *Proc. Middleware*, 2001.

[19] P. Scheuermann, G. Weikum, and P. Zabback. Adaptive load balancing in disk arrays. In *Proc. FODO*, 1993.

[20] S.Ratnasamy, J.M.Hellerstein, and S.Shenker. Range queries over DHTs. Technical Report IRB-TR-03-009, Intel, 2003.

[21] T. Stohr, H. Martens, and E. Rahm. Multi-dimensional database allocation for parallel data warehouses. In *Proc. VLDB*, 2000.

[22] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. SIGCOMM*, 2001.

[23] Tandem Database Group. Nonstop sql, a distributed high-performance, high-reliability implementation of sql. In *Proc. HPTS*, 1987.

[24] D. C. Zilio. *Physical Database Design Decision Algorithms and Concurrent Reorganization for Parallel Database Systems*. PhD thesis, University of Toronto, 1988.